The ITP Tool's Manual*

Manuel Clavel and Miguel Palomino
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
clavel@sip.ucm.es miguelpt@sip.ucm.es

April 12, 2005

Abstract

The ITP tool is an experimental inductive theorem prover for proving properties of Maude equational specifications, i.e., specifications in membership equational logic with an initial algebra semantics. The ITP tool has been written entirely in Maude and is in fact an executable specification of the formal inference system that it implements.

1 Getting started

To run the current version of the ITP you need to have installed the latest version of the Maude system (version 2.1.1) in your computer. You can download the latest version of Maude from the Maude's home page: `http://maude.cs.uiuc.edu`.

The current version of the ITP is distributed as the gzip-tar-file `itp-tool-v`xxx`.tar.gz` (where xxx is the actual version identifier), which is available at `http://maude.sip.ucm.es/itp`. To untar this file, type `tar xvfz itp-tool-v`xxx`.tar.gz`. This command will create a directory `itp-tool-v`xxx with a subdirectory `itp-src` containing the ITP tool's source files:

```
bash>tar xvfz itp-tool-vxxx.tar.gz
itp-tool-vxxx/
itp-tool-vxxx/itp-src/
itp-tool-vxxx/itp-src/unification.maude
itp-tool-vxxx/itp-src/basic.maude
itp-tool-vxxx/itp-src/check.maude
itp-tool-vxxx/itp-src/ext-mod.maude
itp-tool-vxxx/itp-src/ext-term.maude
itp-tool-vxxx/itp-src/itp-tool.maude
bash>
```

2 Understanding the ITP

Mechanical reasoning about functional modules in Maude is supported by the experimental ITP tool, a rewriting-based theorem prover that can be used to prove inductive properties of membership equational specifications. It is written in Maude, and it is itself an executable

*Research supported by Spanish MCYT projects TIC2002-01167 and TIC2003-01000.

specification. A key feature of the ITP is its reflective design, that allows the definition of customized *rewriting strategies* different from Maude's default one; currently, this capability is being used to extend the ITP with decision procedures for arithmetic, lists, and combinations of theories. The ITP is still work in progress: it can work with any module present in Maude's database but not with those introduced in Full Maude; in particular, at present it offers no support for parameterized specifications (although there are plans to include it in the future).

2.1 A first proof

We will use the following specification of lists of integers as our running example.

```
fmod LIST is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .
  op _:_ : Int List -> NeList [ctor] .
  op _++_ : List List -> List .

  var I : Int .
  vars L L' : List .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .
endfm
```

An obvious property the `LIST` specification should satisfy is that concatenation of lists (`_++_`) is associative. To prove it, once the `LIST` module has been added to Maude's database we load the ITP with the instruction `in itp-tool` and initialize its own database with the command `loop init-itp .`; then, the property can be presented to the ITP using the command `goal`:

```
Maude> (goal list-assoc : LIST |- A{L1:List ; L2:List ; L3:List}
             (((L1:List ++ L2:List) ++ L3:List) =
               (L1:List ++ (L2:List ++ L3:List))) .)
```

Here, `list-assoc` is the name of the goal, `LIST` is the module in which it is to be proved, and the symbol `A` (representing $\forall$) precedes a list of universally quantified variables which have to be annotated with their sorts. Due to parsing restrictions it is convenient to be generous in the use of parentheses; in particular, note that they are used to enclose the terms to be proved equal.

The ITP then outputs

```
==================================
list-assoc$0
==================================


|- A{L1:List ; L2:List ; L3:List}((L1:List ++ L2:List)++ L3:List = L1:List ++(
    L2:List ++ L3:List))

++++++++++++++++++++++++++++++++++
```

indicating that the goal has been correctly processed, has been internally labelled as `list-assoc$0`, and is ready to be worked upon.

Now we can try to prove the property by structural induction on the first variable, using the `ind` command.

```
Maude> (ind on L1:List .)
```

The ITP then generates a corresponding subgoal for each operator with codomain `List` that has been declared with the attribute `ctor` (taking subsorts into account), but only shows the one corresponding to the first one, in this case the empty list `[]`:

```
==================================
list-assoc$1.0
==================================


|- A{L2:List ; L3:List}(([]++ L2:List)++ L3:List =[]++(L2:List ++ L3:List))

++++++++++++++++++++++++++++++++++
```

At this point, we can try to automatically prove the subgoal with the command `auto`, that first transforms all variables into fresh constants and then rewrites the terms in both sides of the equality as much as possible by using the equations in the module as rewrite rules.

```
Maude> (auto .)
```

The command succeeds, the subgoal is discharged, and the ITP presents us with the remaining subgoal generated by the induction, the one corresponding to the operator `_:_`.

```
==================================
list-assoc$2.0
==================================


|- A{V0#0:Int ; V0#1:List}((A{L2:List ; L3:List}((V0#1:List ++ L2:List)++
    L3:List = V0#1:List ++(L2:List ++ L3:List)))==>(A{L2:List ; L3:List}(((
    V0#0:Int : V0#1:List)++ L2:List)++ L3:List =(V0#0:Int : V0#1:List)++(
    L2:List ++ L3:List))))

++++++++++++++++++++++++++++++++++
```

Though this output is all but clear, notice that the expression before the arrow `==>` corresponds to the induction hypothesis and how the term `V0#0:Int : V0#1:List` is substituted for `V0#1:List` in the subsequent expression. We can also try to prove this subgoal automatically and, again, the ITP succeeds and this completes the proof.

```
Maude> (auto .)

q.e.d

++++++++++++++++++++++++++++++++++
```

2.2 Constructor and defined memberships and the `ind` command

The behavior of the `ind` command deserves a closer look: how does the ITP generate the inductive subgoals? In the `LIST` module, the operators `[]` and `_:_` are declared to be constructors of the sorts `List` and `NeList`, respectively, by means of the attribute `ctor`; since `NeList` is a subsort of `List`, this automatically makes `_:_` a constructor of `List` as well. Then, when asked to prove a goal by structural induction on a variable of sort `List`, the `ind` command generates two subgoals:

- The first one, that corresponds to the base case, is obtained by replacing the variable with the constant `[]` in the goal.
- The second one, which is somewhat more obscure and corresponds to the inductive step, is built as an implication where the antecedent and the consequent arise from the original goal by replacing the variable with a fresh one in the first case, and with a term constructed using `_:_` in the second.

The situation is a bit more complicated in general and requires the notion of a constructor membership. In Maude, an operator declaration `op` f `:` $s_1 \cdots s_n$ `->` s_0 `[`*AttrS*`]` `.` is logically equivalent to a declaration `op` f `:` $[s_1] \cdots [s_n]$ `->` $[s_0]$ `[`*AttrS*`]` `.` at the kind level and a membership axiom `mb` f`(`x_1`:`s_1`,...,`x_n`:`s_n`)` `:` s_0 `.`. The ITP, in addition, distinguishes those operator declarations that contain the `ctor` attribute from those that do not, and tags the membership axioms associated to the latter with the label `metadata "dfn"`. Hence, the `LIST` module is interpreted by the ITP as:

```
fmod LIST-MB is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> [List] .
  op _:_ : [Int] [List] -> [NeList] .
  op _++_ : [List] [List] -> [List] .

  var I : Int .
  vars L L' : List .

  mb [] : List .
  mb (I : L) : NeList .
  mb (L ++ L') : List [metadata "dfn"] .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .
endfm
```

The *defined memberships* of a functional module are the memberships in the transformed module that are tagged with the label `metadata "dfn"`; the remaining ones are called *constructor memberships*. When the `ind` command is used to reason by structural induction on a variable of sort s, it generates the goals that correspond to the inductive cases from the constructor memberships associated to s and to all its subsorts.

The defined memberships in a specification provide in a handy manner helpful information for the ITP to reason with, but in "good" specifications they *should* actually be redundant and deducible from the rest of the specification (for example, it is clear that the concatenation of

two lists should also be a list). For this situation to actually hold, the equations that define the operations have to thoroughly consider all possibilities so that every term eventually reduces to a canonical form to which a constructor membership applies. This is the sufficient completeness problem: see Section 7.1.

3 A script safari

The essentials of the ITP have already been covered in the previous section: terms are proved equal by reducing them to syntactically equal terms and structural induction is based on the constructor memberships. Here we continue exploring other ITP commands and illustrate how they are used in several scripts.

3.1 Defined memberships revisited

To stress how defined memberships are treated by the ITP, let us extend the `LIST-MB` module with an operator `empty?` that checks whether a list is empty or not. Admittedly, since we already have a sort `NeList` to distinguish nonempty lists, the use of the operator `empty?` in this example is a bit contrived; let us stick with it for the sake of the argument.

```
fmod LIST-MB is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> [List] .
  op _:_ : [Int] [List] -> [NeList] .
  op _++_ : [List] [List] -> [List] .
  op empty? : List -> Bool .

  var I : Int .
  vars L L' : List .

  mb [] : List .
  mb (I : L) : NeList .
  mb (L ++ L') : List [metadata "dfn"] .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .

  eq empty?([]) = true .
  eq empty?(I : L) = false .
endfm
```

The ITP can also be used to prove membership assertions; in particular, we can try to show that the term `empty?(`L`)` is of sort `Bool` for every list L.

```
Maude> (goal empty-bool : LIST-MB |- A{L:List} ((empty?(L:List)) : Bool) .)
```

As expected, since the specification includes the declaration

```
  op empty? : List -> Bool .
```

the goal can be discharged by just using `auto`. What would have happened, however, if we had forgotten to add the equation

```
eq empty?([]) = true .
```

to the specification? Obviously, this would have left undefined `empty?` over the empty list with the undesired consequence that now not all terms of the form `empty?(`L`)` are of sort `Bool`. Nonetheless, the ITP would have still proved it! As mentioned in Section 2.2, this is the sufficient completeness problem: to guarantee that the equations for the defined operators in the specification consider all possible cases. In this situation, the defined membership

```
mb empty?(L) : Bool [metadata "dfn"] .
```

associated to the declaration of `empty?` is *not* redundant and cannot be derived from the rest of the specification.

3.2 The `ctor-split` command

From what we have just seen, a safer approach to specification would then be to declared all defined operators at the kind level and to use the ITP to show that they have the intended typing.

```
op empty? : [List] -> [Bool] .

eq empty?([]) = true .
eq empty?(I : L) = false .
```

Now, if we present to the ITP our desired goal again

```
Maude> (goal empty-bool : LIST-MB |- A{L:List} ((empty?(L:List)) : Bool) .)
```

and try to prove it with the `auto` command, we get the following output:

```
==================================
empty-bool$1.1.1.0
==================================


|- empty?(L*List): Bool

++++++++++++++++++++++++++++++++++
The goal labeled empty-bool$1.1.1.0 is not an identity
```

Since the defined membership is no longer available, the ITP cannot jump to the conclusion that `empty?(L*List)` has sort `Bool`, and since `L*List` is a fresh constant, none of the equations in the specifications can be applied to further reduce the term. To continue with the proof, we apply a case analysis using the `ctor-split` command, that studies what happens for each of the different constructors of the sort `List`. (Induction cannot be applied here because `L*List` is a constant; of course, the `ind` command could have been used before applying `auto` and that would yield a different proof.)

```
Maude> (ctor-split on L*List .)

==================================
empty-bool$1.1.1.1.0
```

```
==================================


|-(L*List =[])==>(empty?(L*List): Bool)

++++++++++++++++++++++++++++++++++
```

The first case corresponds to the empty list and is easily discharged with `auto`.

```
Maude> (auto .)

==================================
empty-bool$1.1.1.2.0
==================================


|- A{V0#0:Int ; V0#1:List}((L*List = V0#0:Int : V0#1:List)==>(empty?(L*List):
    Bool))

++++++++++++++++++++++++++++++++++
```

The second case corresponds to the constructor `_:_` and is also proved with `auto`.

```
Maude> (auto .)

q.e.d

++++++++++++++++++++++++++++++++++
```

Even though, as mentioned earlier, declaring the defined operators at the kind level offers a safer way of writing specifications, we do not encourage this practice because it would imply the necessity of having to prove manually plenty of tedious and trivial details. Of course, the sufficient completeness problem cannot be just swept aside, but to deal with it we suggest instead to use the sufficient completeness analyser that has been integrated with the last release of the ITP (see Section 7.1).

For example, for the version of the `LIST-MB` module that contains the declaration of `empty?` at the sort level, but with the equation `empty?([]) = true` missing, the result would be:

```
Maude> (scc LIST-MB .)

==================================
CTOR-LIST-MB$1.0
==================================


|- true = false

++++++++++++++++++++++++++++++++++
```

Though not very informative, this output is enough to point out that the specification is not sufficiently complete.

3.3 Another induction scheme: `c-ind` and `show-all`

In addition to the `ind` command to reason by induction on the structure of terms, the ITP also offers the `(c-ind on  .)` command to reason by induction over the natural numbers. This command takes a term of sort `Nat` as argument and generates two subgoals from the original goal: one in which the term is assumed to be equal to `0` and another one in which it states that the goal holds for `N` assuming that it holds for values less than `N`. To illustrate its use, let us extend the `LIST-MB` with an operation to determine the length of a list.

```
  op length : List -> Nat .

  eq length([]) = 0 .
  eq length(I : L) = length(L) + 1 .
```

Now, the associativity of the operator `_++_` can alternatively be proved by using `c-ind`:

```
Maude> (goal list-assoc : LIST-MB |- A{L1:List ; L2:List ; L3:List}
             (((L1:List ++ L2:List) ++ L3:List) =
               (L1:List ++ (L2:List ++ L3:List))) .)
...

Maude> (c-ind on (length(L1:List)) .)

==================================
list-assoc$1.0
==================================


|- A{L1:List ; L2:List ; L3:List}((length(L1:List)= 0)==>((L1:List ++
    L2:List)++ L3:List = L1:List ++(L2:List ++ L3:List)))

++++++++++++++++++++++++++++++++++
```

The first subgoal simply assumes that `length(L1:List)` is `0`. Using `auto` seems to have no effect except for the transformation of the variables into constants.

```
Maude> (auto .)

==================================
list-assoc$1.1.1.1.1.1.0
==================================


|-(L1*List ++ L2*List)++ L3*List = L1*List ++(L2*List ++ L3*List)

++++++++++++++++++++++++++++++++++
The goal labeled list-assoc$1.1.1.1.1.1.0 is not an identity
```

To try to understand what is going on we can use the `show-all` command, which outputs the functional module the ITP is currently reasoning about (see Section 4).

```
Maude> (show-all .)
```

The resulting module is identical to the original `LIST-MB` (with `length`) except for the additional declaration of the new constants `L1*`, `L2*`, and `L3*`, and the equation `length(L1*)`

= 0. Obviously, this additional information is not enough to further reduce any of the terms in the goal.

To proceed with the proof we must supply the ITP with additional guidelines to follow and what turns out to be useful in this case is to make a case analysis using `ctor-split`.

```
Maude> (ctor-split on L1*List .)

==================================
list-assoc$1.1.1.1.1.1.1.0
==================================


|-(L1*List =[])==>((L1*List ++ L2*List)++ L3*List = L1*List ++(L2*List ++
    L3*List))

++++++++++++++++++++++++++++++++++
```

Now we can try to prove the goal assuming that `L1*List` is the empty list, and `auto` easily succeeds.

```
Maude> (auto .)

==================================
list-assoc$1.1.1.1.1.1.2.0
==================================


|- A{V1#0:Int ; V1#1:List}((L1*List = V1#0:Int : V1#1:List)==>((L1*List ++
    L2*List)++ L3*List = L1*List ++(L2*List ++ L3*List)))

++++++++++++++++++++++++++++++++++
```

The remaining goal deserves closer attention. The list `L1*List` is now built using `_:_` and is not clear at all why the goal should be true. But recall that this subgoal has arisen while we are trying to prove the first subgoal generated by `c-ind`, that is, the one in which `length(L1*List)` is `0`. And this is in contradiction with `L1*List` being constructed with `_:_` (all such terms must have length at least equal to 1, according to the equations for `length`): `auto` notices this contradiction and automatically discharges the subgoal.

```
Maude> (auto .)

==================================
list-assoc$2.0
==================================


|- A{V0#0:Nat}((A{L1:List ; L2:List ; L3:List}((length(L1:List)< V0#0:Nat =
    true)==>((L1:List ++ L2:List)++ L3:List = L1:List ++(L2:List ++
    L3:List))))==>(A{L1:List ; L2:List ; L3:List}((length(L1:List)=
    V0#0:Nat)==>((L1:List ++ L2:List)++ L3:List = L1:List ++(L2:List ++
    L3:List)))))

++++++++++++++++++++++++++++++++++
```

The proof of this second subgoal generated by `c-ind` proceeds exactly like that for the first one. After applying `auto`, it is necessary to make a case analysis whose subcases can be discharged with `auto`. (This time, it is the proof of the first case which succeeds by contradiction.)

```
Maude> (auto .)
       (ctor-split on L1*List .)
       (auto .)
       (auto .)
```

In this example, the script produced by `c-ind` is certainly more cumbersome than the one for `ind`. However, there are proofs (e.g., in the merge sort algorithm) where `c-ind` gives rise to proofs that cannot easily be done by structural induction.

3.4 A more complex example: `lem` and `split`

Let us extend now the original `LIST` specification with an operation `insertion-sort` that sorts a list using the insertion sort algorithm, and a Boolean operation `sorted?` that checks if a list is sorted.

```
fmod LIST is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .
  op _:_ : Int List -> NeList [ctor] .

  op _++_ : List List -> List .
  op insertion-sort : List -> List .
  op insert-list : List Int -> List .
  op sorted? : List -> Bool .

  vars I I' : Int .
  vars L L' : List .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .

  eq insertion-sort([]) = [] .
  eq insertion-sort(I : L) = insert-list(insertion-sort(L), I) .
  eq insert-list([], I') = I' : [] .
  ceq insert-list(I : L, I') = I' : I : L if I' <= I .
  ceq insert-list(I : L, I') = I : insert-list(L, I') if I' > I .

  eq sorted?([]) = true .
  eq sorted?(I : []) = true .
  ceq sorted?(I : I' : L) = false if I' < I .
  ceq sorted?(I : I' : L) = sorted?(I' : L) if I <= I' .
endfm
```

We want the list L returned by a call to `insertion-sort` to be actually ordered so that `sorted?(`L`) = true`, and we set ourselves to prove it. (For the operation `insertion-sort`

to be well-defined it would also be necessary to prove that L is a permutation of the original list, that is, that the elements in L are exactly the same as in the original list and with the same multiplicity: we leave this proof as an exercise to the reader.)

```
Maude> (goal SORTED : LIST |- A{L:List}
             ((sorted?(insertion-sort(L:List))) = (true)) .)
```

The proof proceeds by structural induction on `L`. For the empty list, the result follows immediately using `auto`; in the inductive step, however, after applying `auto` we arrive to:

```
==================================
SORTED$2.1.1.1.1.1.0
==================================


|- sorted?(insert-list(insertion-sort(V0#1*List),V0#0*Int))= true

++++++++++++++++++++++++++++++++++
The goal labeled SORTED$2.1.1.1.1.1.0 is not an identity
```

At this point, the ITP can no longer reduce the term and gets stuck.

Let us think about what remains to be proved. By the induction hypothesis, the list `insertion-sort(V0#1*List)` is sorted and, since we believe the equations for `insert-list` are correct, we would expect `insert-list(insertion-sort(V0#1*List),V0#0*Int)` to be sorted as well, even though it can be reduced no further. What we need is precisely this, an auxiliary result that states that inserting a new element into an ordered list using `insert-list` results in another ordered list. Such a lemma can be introduced in the ITP with the `lem` command and has to be proved for the reasoning to be sound. In the current ITP's version, the lemma actually has to be proved before it can be used; this restriction will probably be removed in future versions.

```
Maude> (lem sort1 : A{I:Int ; L:List}
            ((sorted?(L:List)) = (true) =>
             (sorted?(insert-list(L:List,I:Int))) = (true)) .)

==================================
sort1$0
==================================


|- A{I:Int ; L:List}((sorted?(L:List)= true)==>(sorted?(insert-list(L:List,
    I:Int))= true))

++++++++++++++++++++++++++++++++++
```

The syntax for lemmas is the same as for goals, except for the fact that no module is specified. Note that the statement to be proved in this case is not just an equality, but an implication. Also, the name of the goal has been changed to `sort1$0` to reflect that the lemma has become the current goal.

Again, we can think of proving the lemma with the `ind` command and, as happened for the main goal, `auto` discharges the base case but merely transforms the inductive step into

```
==================================
sort1$2.1.1.1.1.1.1.1.1.0
```

```
==================================


|- sorted?(insert-list(V1#0*Int : V1#1*List,I*Int))= true

++++++++++++++++++++++++++++++++++
The goal labeled sort1$2.1.1.1.1.1.1.1.1.0 is not an identity
```

If we take a look at the equations for `insert-list`, the reason why this term cannot be reduced becomes apparent: the two equations that might apply are conditional and depend on whether `I*Int <= V1#0*Int` or `I*Int > V1#0*Int`. Therefore, we use the `split` command with the term `I*Int <= V1#0*Int`, to generate a subgoal in which the term is taken to be `true` and another one in which is considered to be `false`; discharging both goals will prove that from which they arose.

```
Maude> (split on (I*Int <= V1#0*Int) .)

==================================
sort1$2.1.1.1.1.1.1.1.1.0
==================================


|- sorted?(insert-list(V1#0*Int : V1#1*List,I*Int))= true

++++++++++++++++++++++++++++++++++
```

At first sight, nothing has changed, and that is true, but only as far as the goal is concerned: internally, the module the ITP is using to prove the goal has been extended with the additional equation

```
  eq I*Int <= V1#0*Int = true
```

which can be checked by the user with the `show-all` command.

So let us resume the proof. The current goal is discharged with `auto`, which leaves us with another one which is apparently the same (but now, under the assumption that `I*Int <= V1#0*Int = false`). However, if we now apply the same command `auto`, we end up with

```
==================================
sort1$2.1.1.1.1.1.1.1.1.2.1.1.0
==================================


|- sorted?(V1#0*Int : insert-list(V1#1*List,I*Int))= true

++++++++++++++++++++++++++++++++++
The goal labeled sort1$2.1.1.1.1.1.1.1.1.2.1.1.0 is not an identity
```

Under the current assumption that `I*Int <= V1#0*Int = false` and the induction hypothesis that `insert-list(V1#0*Int : V1#1*List,I*Int)` is ordered, it is clear that the equality holds. What we need is yet another lemma that makes this observation a general and explicit statement.

```
Maude> (lem sort2 : A{I:Int ; I':Int ; L:List}
            ((sorted?(I:Int : L:List)) = (true) & (I:Int <= I':Int) = (true) =>
             (sorted?(I:Int : insert-list(L:List,I':Int))) = (true)) .)
```

Note that the symbol `&` is used to separate conjunctions in the antecedent of the implication.

The proof of this lemma proceeds along lines that should be familiar by now; no new auxiliary results are needed and we just present the necessary commands: we encourage the reader to try to obtain them by himself.

```
Maude> (ind on L:List .)
       (auto .)
       (auto .)
       (split on (I'*Int <= V2#0*Int) .)
       (auto .)
       (auto .)
       (split on (I*Int <= V2#0*Int) .)
       (auto .)
       (auto .)
```

Once the last `auto` in the script above has been executed, the ITP prompts us with the goal labelled `sort1$2.1.1.1.1.1.1.1.1.2.1.1.0`, which is the point where the proof was interrupted to introduce the lemma `sort2`; with this result at our disposal we can discharge it with `auto`. This brings forward the goal `SORTED$2.1.1.1.1.1.0` and, similarly, the ITP can finally prove it with `auto` by using `sort1` and complete the proof.

4 The ITP's module database

5 ITP commands

5.1 (goal *Label* : *Entailment* .)

Syntax `op goal_:_. : Token Goal -> Input .`

Summary It introduces the goal to be proved in the ITP.

Details This command introduces a goal, named *Label*, to be proved with the ITP. The expression *Entailment* has the form *IdMod* `|- A{`$x_1 : s1, \ldots, x_n : s_n$`}` *expression*, with *IdMod* the name of the module the goal is about, $x_1, \ldots, x_n$ variables, and *expression* a Horn clause built with `&` and `=>`.

5.2 (lem *Label* : *LEntailment* .)

Syntax `op lem_:_. : Token UserFormula -> Input .`

Summary It introduces an auxiliary result to be proved in the ITP.

Details This command introduces a lemma, named *Label*, to be proved with the ITP. The expression *LEntailment* has the form `|- A{`$x_1 : s1, \ldots, x_n : s_n$`}` *expression*, with $x_1, \ldots, x_n$ variables and *expression* a Horn clause built with `&` and `=>`.

5.3 (auto .)

Syntax `op auto . : -> Input .`

Summary It tries to automatically prove the current goal.

Details It first transforms all the variables into fresh constants and then reduces the terms in the goal as much as possible by using the equations in the module as rewrite rules. When necessary, decision procedures are applied to check if the conditions of conditional equations are satisfied.

5.4 (ind on *Var* .)

Syntax `op ind on_. : Token -> Input .`

Summary It generates the set of goals that correspond to the inductive cases necessary to prove the goal by structural induction on the sort of the variable *Var*.

Details The goals are generated from the constructor memberships for the sort of *Var* and all its subsorts. Base cases correspond to unconditional memberships of the form `mb` $t : s$., and give rise to new subgoals by replacing the *Var* in the current goal with t. The inductive steps correspond to conditional memberships.

5.5 (c-ind on Nat-*Term* .)

Syntax `op c-ind on (_). : Bubble -> Input .`

Summary It proceeds to prove the goal by induction on the value of `Nat-`*Term*.

Details It generates two subgoals from the current one. The first one consists in the original goal assuming that the value of `Nat-`*Term* is `0`. The second subgoal states that if the original one holds when the value of `Nat-`*Term* is less that `N`, then it also holds when the value of `Nat-`*Term* is `N`.

5.6 (ctor-split on *Const* .)

Syntax `op ctor-split on_. : Token -> Input .`

Summary It generates the set of goals that correspond to a case analysis over the structure of the terms with the sort of *Const*.

Details The goals are generated from the constructor memberships for the sort of *Const* and all its subsorts, in the same way as for `ind`, but no induction hypothesis is generated.

5.7 (scc *IdMod* .)

Syntax `op scc_. : Token -> Input .`

Summary It generates the set of goals corresponding to the proof obligations which, if discharged, ensure sufficient completeness of the *IdMod* functional module.

Details It first calls on the functional module named *IdMod* the function `checkCompleteness`, which implements the SCC's sufficient completeness analizer. Then, it converts the resulting proof obligations into a set of goals, which are all associated with the constructor submodule of *IdMod*. Finally, it eliminates from the state of the proof those goals that can be proved automatically using the ITP's `auto` command.

5.8 `(show-all .)`

Syntax `op show-all . : -> Input .`

Summary It outputs the active module in the ITP's module database.

Details

5.9 `(split on (Bool-`*Term*`) .)`

Syntax `op split on (_). : Bubble -> Input .`

Summary It splits the current goal in two: one in which `Bool-`*Term* is assumed to be `true` and another one in which it is assumed to be `false`.

Details

6 ITP projects

6.1 ASIP: algebraic semantics of imperative programs

The ASIP project is based on J. Goguen's seminal work on algebraic semantics of imperative programs. It aims to provide a version of the ITP that may be used to formally specify and verify software.

7 ITP related tools

7.1 SCC: a sufficient completeness checker

SCC [1] is an experimental tool for checking the *sufficient completeness* of partial specifications written in Maude. Sufficient completeness is the property that operations are defined on all valid inputs. It is an important property both for developers of specifications, to check that they have not missed a case while defining the operations, and to inductive theorem provers, to check the soundness of a proposed induction scheme. The SCC tool has been written in Maude and relies on Maude's reflective capabilities and the ITP tool. The SCC tool is included in the latest distribution of the ITP tool, and it can be executed both in stand-alone mode and through appropriate commands during an ITP session.

A Glossary

Constructor membership. The constructor memberships of a given module named *IdMod* are:

- the (conditional) membership axioms in *IdMod* that are not declared with the `defn` attribute; and
- membership axioms `mb` f`(`x_1`:`s_1`,...,`x_n`:`s_n`)` `:` s_0 `.`, for every operator declaration `op` f `:` $s_1 \cdots s_n$ `->` s_0 `[`*AttrS*`]` `.` in *IdMod* such that *AttrS* contains the `ctor` attribute.

Constructor submodule. The constructor submodule of a given module named *IdMod* is the submodule of *IdMod* that contains:

- the many-kinded signature in *IdMod*;
- all the equations in *IdMod*;
- all the subsort declarations in *IdMod*; and
- only the constructor memberships in *IdMod*.

Defined membership. The set of defined memberships of a given module *IdMod* is given by:

- the (conditional) membership axioms in *IdMod* that are declared with the `defn` attribute; and
- the membership axioms `mb` f`(`x_1`:`s_1`,...,`x_n`:`s_n`) :` s_0 `.` for every operator declaration `op` f `:` $s_1 \cdots s_n$ `->` s_0 `[`*AttrS*`]` `.` in *IdMod* such that *AttrS does not* contain the `ctor` attribute.

Subsort declaration. From a mathematical point of view, a subsort declaration $s < s'$ is logically equivalent to a membership axiom $(\forall x)\, x : s'$ **if** $x : s$. In Maude, however, since kinds cannot be explicitly declared, subsort declarations cannot be desugared in this way.

Sufficient completeness. A module named *IdMod* is sufficiently complete when its operations are defined on all the terms of the appropriate sorts, i.e., they always return terms of the expected sort when called on terms of the appropriate sorts.

More formally, a module named *IdMod* is sufficiently complete relative to its constructor memberships if and only if the unique homomorphism between the algebra of terms of its constructor submodule and its own algebra of terms is an isomorphism. See [2] for more details.

References

[1] J. Hendrix. The SCC tool's home page. `http://maude.cs.uiuc.edu/tools/scc/`, 2005.

[2] J. Hendrix, M. Clavel, and J. Meseguer. A sufficient completeness reasoning tool for partial specifications. To appear in *RTA'05: Proceedings of the 15th International Conference. Nara, Japan, April*, 2005.